SCHEDULE FAILURE ANALYSIS WITHIN THE HORIZON SIMULATION FRAMEWORK

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Aerospace Engineering

by

Ian Lunsford

June 2016

COMMITTEE MEMBERSHIP

TITLE:	Schedule Failure Analysis within the Horizon Simulation Framework
AUTHOR:	Ian Lunsford
DATE SUBMITTED:	June 2016
COMMITTEE CHAIR:	Eric Mehiel, Ph.D. Professor of Aerospace Engineering
COMMITTEE MEMBER:	Franz Kurfess, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Kira Abercromby, Ph.D. Associate Professor of Aerospace Engineering
COMMITTEE MEMBER:	Ricardo Tubio-Pardavila, M.S. Lecturer of Aerospace Engineering

ABSTRACT

Schedule Failure Analysis within the Horizon Simulation Framework

Ian Lunsford

System design is an inherently expensive and time consuming process. Engineers are constantly tasked to investigate new solutions for various programs. Model-based systems engineering (MBSE) is an up and coming successful method used to reduce the time spent during the design process. By utilizing simulations, model-based systems engineering can verify high-level system requirements quickly and at low cost early in the design process. The Horizon Simulation Framework, or HSF, provides the capability of simulating a system and verifying the system performance. This paper outlines an improvement to the Horizon Simulation Framework by providing information to the user regarding schedule failures due to subsystem failures and constraint violations. Using the C# language, constraint violation rates and subsystem failure rates are organized by magnitude and written to .csv files. Also, proper subsystem failure and constraint violation checking orders were stored for HSF to use as new evaluation sequences. The functionalities of the systemEval framework were verified by five test cases. The output information can be used for the user to improve their system and possibly reduce the total run-time of the Horizon Simulation Framework.

ACKNOWLEDGMENTS

I would like to thank Dr. Eric Mehiel for his consistent guidance throughout the entire development of this Thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. Introduction

1.1 Horizon Simulation Framework Overview

The Horizon Simulation Framework, or HSF, is a model-based systems engineering framework complied from C# source code into a command line program. Utilizing simulations, users can input a model and an environment into HSF to verify and validate high-level system requirements. The systems engineering process is commonly outlined in the Systems Engineering 'V' as seen in Figure 1.[11] The effectiveness of HSF resides within the System Verification and Validation Phase of the 'V'. By implementing model-based systems engineering, or MBSE, verification and validation can be achieved faster and at a lower cost.[4] HSF emulates the MBSE method by simulating conceptually designed systems to confirm their ability to meet system level requirements in an operational environment. Thus, HSF provides the capability to reduce the costs and length of any conceptual design process.

Currently, there have been two systems successfully developed and simulated within HSF. One, being a weather observation satellite named Aeolus. The other successful system was a thermal-seeking aircraft glider.[15] Both had their system level requirements validated and verified through generating an operational system schedule. A schedule is a simulation produced by HSF containing large amounts of information regarding the system within the specified environment. Creating operational schedules is a core function of HSF which allows a model to be evaluated.

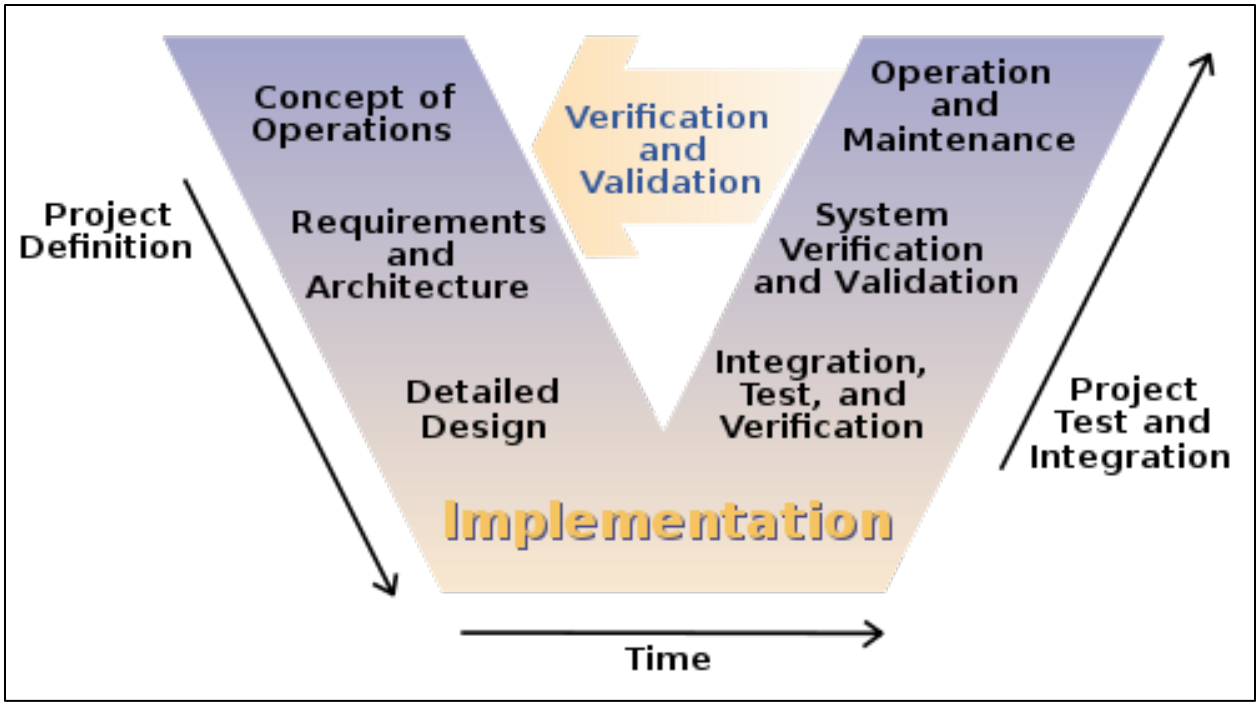

Figure 1: Systems Engineering 'V'[12]

In order to validate system requirements, HSF must produce all possible schedules for a system during a simulation. This is executed within HSF's Scheduler. The recursive functionality of the Scheduler is known as the Big Dumb Exhaustive Search Algorithm, or BDESA. Those produced schedules are recursively scheduled through an algorithm that allocates constrained resources with respect to a cost function. The results of BDESA are all possible schedules of a simulated system. With that information, the user can validate and verify whether system level requirements were met.

Most of the components involved with HSF are defined by the user. Within HSF, resides the Scheduler and the user defined System Model. The inputs are the user defined Simulation Parameters and the System Parameters. After HSF has completely run, the outputs are the Final Schedule with State Data. The high-level structure of HSF can be seen in Figure 2.

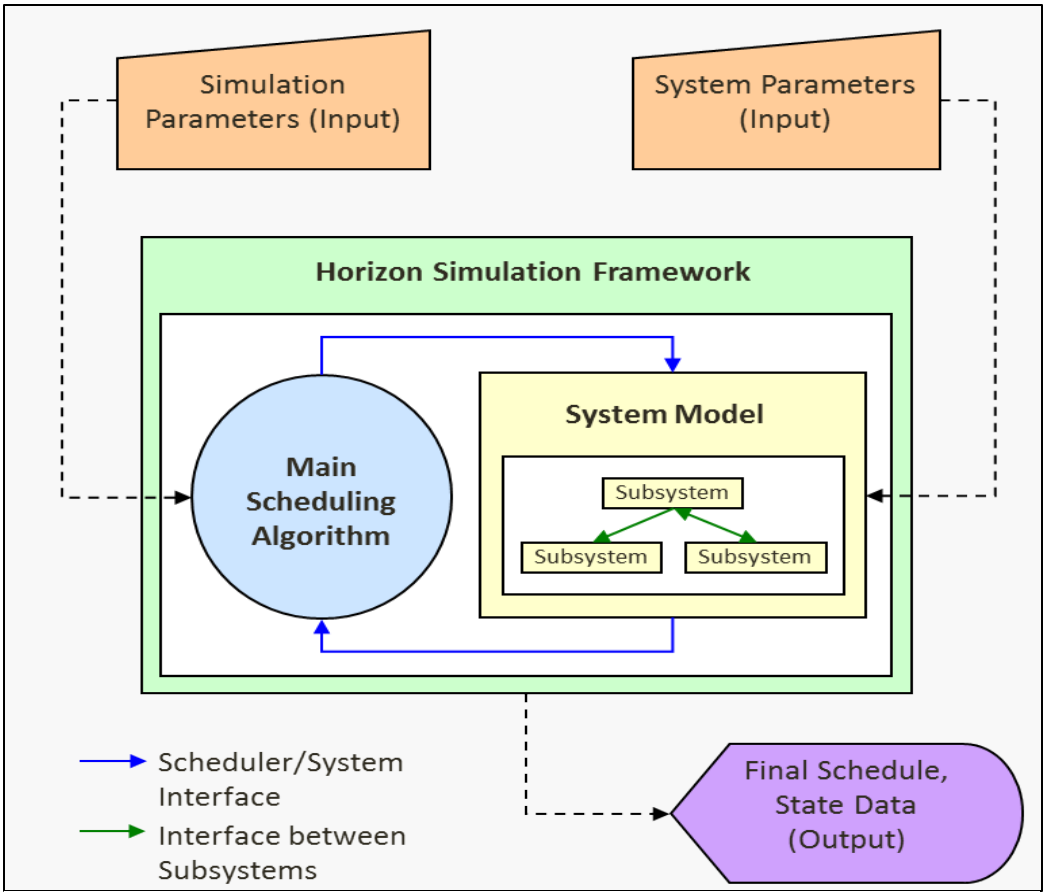

Figure 2: HSF Structure[14]

Within each schedule reside three fundamental scheduling elements: task, state, and event.[9] A task can be considered as the objective of each simulated time step. A task has a target, which is simply a location, a performance characteristic, and the type of action required to perform the task. A state is a vector storage mechanism which contains all information about the system over time. Lastly, an event is task to be performed and the state where the data is recorded. These elements are used to determine whether system level requirements are met in each schedule. Before creating a schedule, the subsystem failures and constraint violations of a system must be checked against all subsystems which include their dependencies.

Constraints, subsystems, and dependencies define the fundamental elements of a system. A constraint is a restriction linked to a subsystem and is compared to the subsystem states. A subsystem is an element which creates state data and affects the ability to perform tasks. Finally, a dependency is the specific relationship between subsystems. Subsystems require their dependencies to be able to perform tasks. When attempting to create schedules, all the system's subsystems are compared to their respective constraints while considering the ability of each subsystem's dependencies to perform tasks. Whenever a subsystem fails or a constraint is violation, the schedule fails.

Schedule failure can happen in any environment and for any system. A schedule could fail because of a subsystem failure or a constraint violation. Using the weather observation satellite, Aeolus, as an example, some instances of schedule failure can be described. If Aeolus must slew to achieve proper orientation for capturing images, there may be a need for adequate time for the ADCS subsystem to slew the satellite. When there is not enough time to slew, the ADCS subsystem fails. This would cause the entire schedule to fail. Another instance could be if Aeolus' power requirement is constrained, there may be a need for the depth of discharge to be below a specific value. When that requirement state data violates the constraint, the schedule fails. The instances of failure of a schedule can apply to a subsystem or constraint in any system, not just a satellite or an aircraft glider.

If no subsystems failed and no constraints are violated, a schedule is created. Then, another schedule is attempted to be produced. Once all possible schedules are created, HSF has finished processing. If a subsystem fails or a constraint is violated, the schedule fails and is not produced. To ease the user's endeavors determining the best schedule, HSF allows the user to weight tasks. Thus, a produced schedule is valued with respect to the weight of tasks completed.

With the most valuable schedules produced, system level requirements can be determined to be verified and validated. The user is presented the best possible schedules relative to their preferred task execution. The model-based systems engineering method provides the system level requirements information quickly and at a low cost. Once HSF has been run, the user has either verified and validated their requirements or knows that their systems will not meet the requirements. The tool described in this paper provides the user with more information regarding subsystem failures, constraint violations, and the capability for the Scheduler to improve the run-time of HSF.

1.2 Problem

The Horizon Simulation Framework previously had two issues. One issue was that it did not record log data regarding failed schedules. Without information of failed schedules, the user was less able to determine issues within their system. System requirements were still able to validated and verified, but user could not know if their system could be modified and improved. Without attaining recorded log data of failed schedules, HSF functionality was much more limited.

The other issue was that HSF did not consider processing effectiveness when checking for subsystem failures and constraint violations. Which in result, HSF did not follow its fail-fast philosophy. The checking of constraints and subsystems is known as the Constraint Cascade, and can be seen in Figure 3 and Figure 4. When checking if subsystems failed or constraints met requirements, the Scheduler would check all constraints and all subsystems even if other constraints were violated more often. Also, even if multiple dependent subsystems were failing or violating constraints more often, HSF would not consider that information. Thus, the requirements validation process took longer than necessary and in nature, was inefficient.

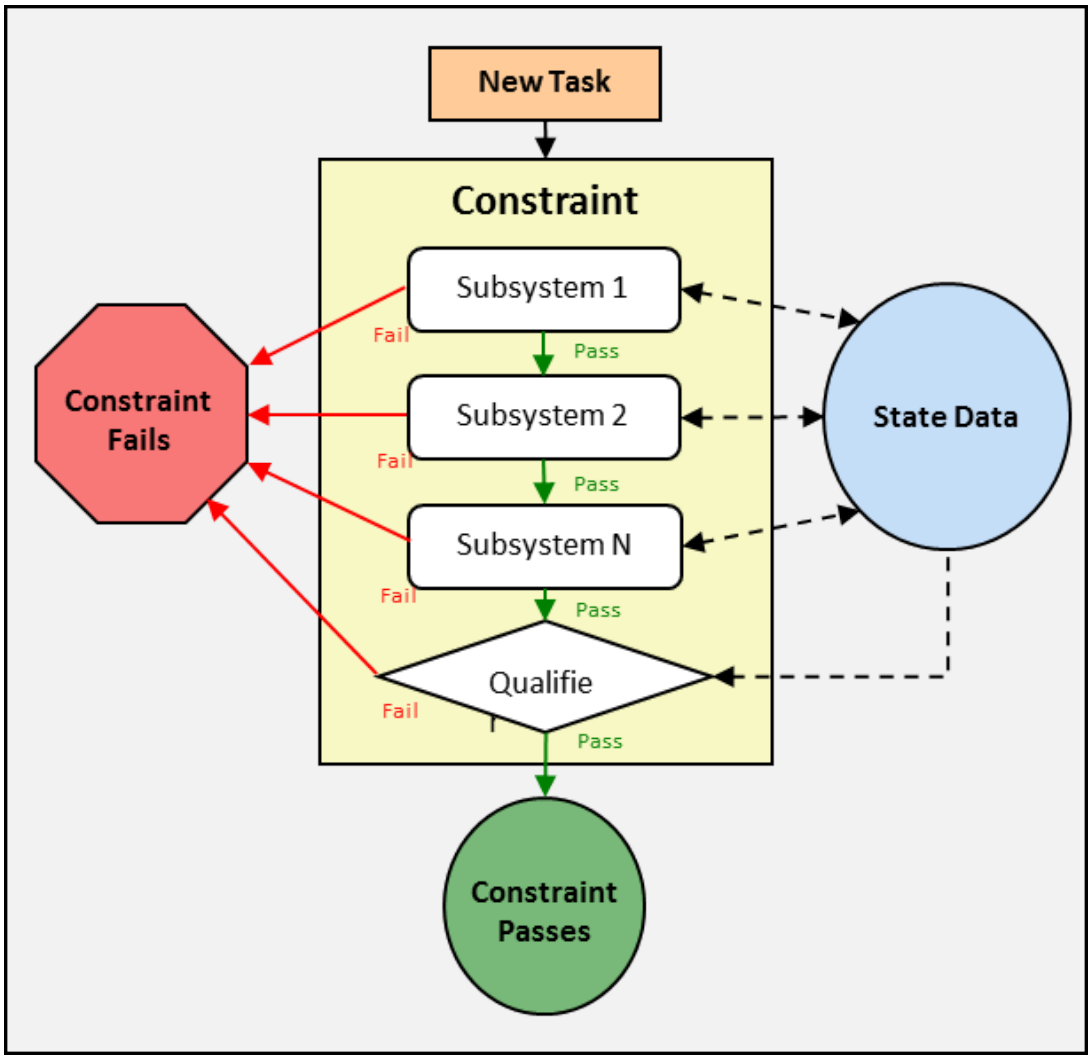

Figure 3: Subsystem Checking

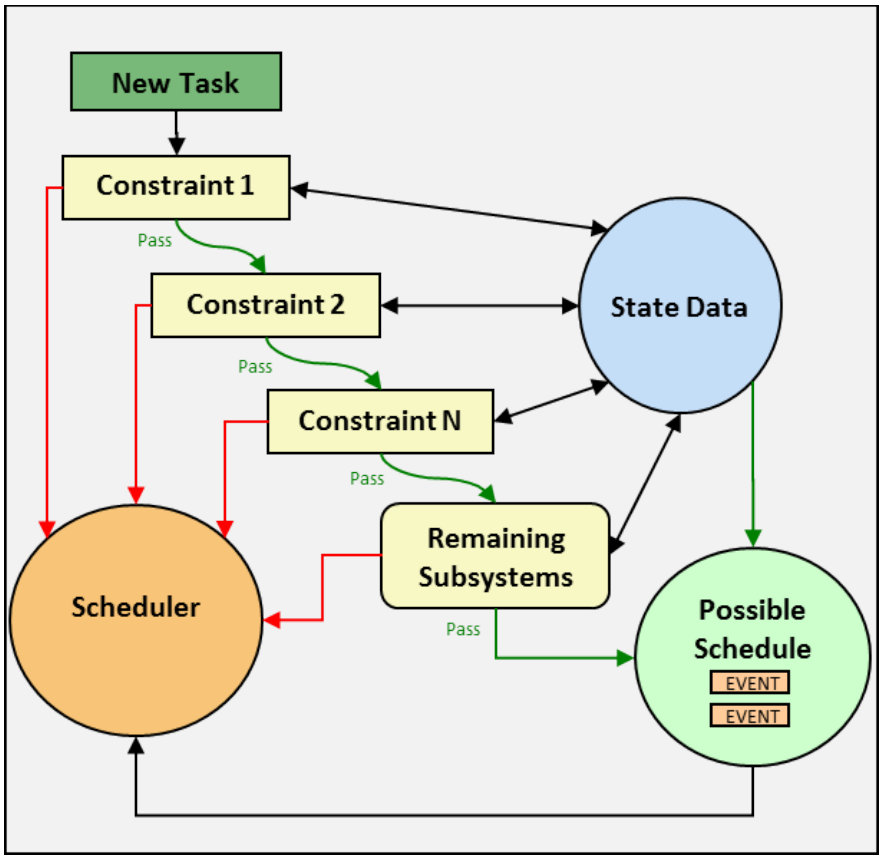

Figure 4: Constraint Cascade

1.3 Paper Overview

Short for system evaluation, the systemEval namespace provides more functionality to HSF. This functionality is described in further detail throughout this paper. First, the problems are expanded upon and the reasoning behind solutions is defined. Then, researched methods of core tool functions are described with decisions for their inclusion or exclusion from the tool. These methods include Bayesian inference and Markov decision processes, both in reference to unsupervised machine learning. After that, the systemEval software test cases are described thoroughly. Next, the results from the test cases of different subsystem trees with varying dependencies using the systemEval namespace is presented and analyzed. Lastly, the

contributions to HSF with the systemEval namespace are concluded and suggestions for future work within the tool are outlined.

1.4 HSF Tool Improvement

As described earlier, HSF can be used to verify and validate high-level system requirements. This is accomplished with assistance from HSF's Scheduler function which creates a large amount of schedules that verify performed tasks specified by the user. In order to consolidate the results, the user specifies the importance of each task relative to each other. This specification allows HSF to determine and output a reasonable number of successful schedules that the user should evaluate. In the essence of practicality, the philosophy of HSF is to fail-fast. This philosophy is based on finding subsystem failures or constraint violations as soon as possible. By failing fast, the run-time of the framework is reduced which provides the user with results sooner. The tool, systemEval, was developed within C# to implement improving HSF's fail-fast philosophy as well as providing information of failing subsystems and violating constraints to the user.

Within HSF, systemEval analyzes failing subsystems and violating constraints and records their rates of occurrence. The logged rates of occurrence aid HSF users in deducing the reasons of failure or violation for each constraint and subsystem. The logged rates of occurrence information are all recorded within systemEval after HSF has processed. To provide the user with a faster processing option, systemEval has two modes: Run Time and Post Process. Where, both modes are within the LogAnalyzer class and calculate the logged rates of occurrence. However, in Post Process mode the results are outputted to a comma delimited .csv file for the user to analyze. Thus, one mode favors more information for the user as the other favors improving run-time of the framework.

In either mode, LogAnalyzer can input subsystem dependencies. Whenever a subsystem has multiple dependencies, LogAnalyzer presents the proper order to check the dependencies for subsystem failure and constraint violation. The proper orders are organized by the highest logged rates of occurrence. With the new information of processing orders, the user or HSF Scheduler can modify the system to fail-faster which in result, would reduce HSF run-time.

In this paper, the systemEval namespace is demonstrated on four different example subsystem trees. There are independent and multiple dependent subsystems throughout the example subsystem trees. The systemEval namespace was demonstrated on the trees in Run Time and Post Process mode. In Post Process mode, the results are presented with organized tables. In whole, the systemEval namespace provides more information for the HSF user and the HSF Scheduler, which improves the overall effectiveness of the Horizon Simulation Framework.

1.5 Research and Literature

In order to determine better process execution, HSF had to consider multiple methods of probability determination and machine learning. Two common methods that were considered for probability determination are a priori and a posteriori probability measurements. These two methods are outlined in multiple references including *Simulation-Based Algorithms for MDPs* and *Decision Making Under Uncertainty – Statistical Decision Theory*.[2,8] These papers' contents and influences are thoroughly described later in the paper.

Another important method that could be used within the tool is machine learning. Today, many machine learning methods have been successfully implemented. Furthermore, research and theories of different methods are rapidly growing. Since, machine learning could be such a powerful function within this tool, a lot of the research done was focused on this topic.

Two main methods of machine learning were specially considered, supervised and unsupervised. Due to the inability to properly infer upon the results of systemEval before they were produced, unsupervised learning was chosen to be the most practical application. Within unsupervised learning, many various methods of machine learning exist. Through much research, a few stood out to be more successful than others. Those methods were Gaussian, Markov, and Bayesian. Although all these methods were deeply considered, simpler methods were chosen to be implemented for the first iteration of log analytics. In future work, machine learning could be very useful since HSF should change the checking order of subsystems and constraints during run-time.

1.6 Background

The Horizon Simulation Framework was built as a utility to the growing field of model-based systems engineering (MBSE). Today, many entities are utilizing the benefits of MBSE to reduce costs, reduce the time of the design process, and improve system performance, which in result improves the overall health of programs.[8] The idea of model-based systems engineering revolves around accurately simulating systems to gather data of their performance. With that ability, problems within designs are determined early, thus reducing time and cost.[12] For example, the conceptual designs of a satellite can be simulated and their ability to perform tasks can be measured. If that is done accurately, all conceptual designs with issues can be identified and filtered out. This would reduce the amount of tasks, time, and costs to output a robust satellite conceptual design. With quick results, more system level requirements can be verified and validated. Thus, allowing more iterations of the conceptual design process which in result improves the system performance. This example is one of the many reasons why MBSE is growing so quickly today.

2. Tool Design

2.1 Deliverables

The final product of the systemEval namespace is a framework written entirely in C# with functionalities that yield specific information. The framework provides the ability to read-in .xml files containing assets, subsystems, state variables, dependencies, and constraints. The outputs of systemEval are logged data of failed schedules, rates of subsystem failures and constraint violations, proper subsystem and constraint checking orders, and if user-specified, .csv files. Providing the many deliverables of the systemEval namespace will create functionalities that improve HSF in whole.

2.2 Objectives

The following objectives determined the capabilities of the systemEval namespace:

- *Readability*: The framework should be thoroughly documented throughout the source code. This is to ensure an understandable logic for future work.
- *Practicality*: To uphold the fail-fast philosophy of HSF, the tool should be efficient in a practical sense. Therefore, efficient code logic in the favor of processing time is preferred. Utilizing C# methods are preferred, since they are mostly designed for fast processing.
- *Functionality*: The tool should be easy to use for the user, while providing multiple capabilities. This is to improve the effectiveness of HSF as a whole and to provide the user with more information of their system.

2.3 Requirements

The systemEval namespace provides the following features:

- *Failing Subsystem Rates*: To further the understanding a simulated system's capabilities, the systemEval namespace shall record when subsystems fail and their rate of failure for all failed Schedules.
- *Violating Constraint Rates*: To additionally improve the understanding of a simulated system's abilities, the systemEval namespace shall record when state variables violate their constraints and each constraint's violation rate for all failed Schedules.
- *Proper Processing Order*: The tool shall output proper processing orders of checking subsystems and constraints. The order shall be based on the rates of subsystem failures and constraint violations.
- *Logger*: The tool shall include a logger of subsystem failures and constraint violations for all failed schedules. The Logger shall record log data of the following information: asset names, failing subsystem names, failing task names, failing target names, violating constraint names, violating state variable names, the value of state variable constraint violations, and simulation time information.
- *Run Time Mode and Post Process Mode*: To provide the user with the option of quick results, systemEval shall include two modes: Run Time mode and Post Process mode. Where both Run Time Mode and Post Process mode calculate all subsystem failure rates and constraint violation rates. However, Post Process mode outputs a comma delimited .csv file of the subsystem failure rates and constraint violation rates with other relative information.

- *Comma Delimited .csv File Outputs*: All data written by systemEval shall be in the form of comma delimited .csv files.

2.4 Constraints

In order to be effective, systemEval avoids:

- *Complexity*: Since practicality and readability are objectives, systemEval should avoid complexity. Often complex methods can sacrifice effectiveness and efficiency. Also, to keep HSF's fail-fast philosophy, complex methods that would decrease run-time should be minimized. In addition, complexity should be avoided to ease the understanding of the systemEval namespace in future work.
- *Redundancies:* Also catering to practicality and readability, any redundancies should be avoided. Any code that is not necessary can reduce run-time, which is impractical. In the essence of readability, redundant code should be avoided since it can confuse anyone who is trying to understand how systemEval works. Thus, C# methods are recommended again due to their ease of understanding and efficiency in run-time.

3. Tool Architecture and Logic

3.1 HSF Codebase

Within HSF and for each mode, there are two independent inputs that are utilized by systemEval. One resides within a .xml file and the other is from a failed Schedule. Inside the .xml file is the system model, which contains each asset, subsystems, state variables, dependencies, and constraints. While the failed Schedule, contains a plethora of information which constructs the log data of each Schedule. Only the log data input varies for both modes of systemEval. The .xml file is inputted into HSF where the log data inputs are outputted by HSF and then inputted into systemEval specifically.

3.2 Model .xml Input

The model .xml file is defined with a specific format. Within the file are elements which contain information regarding themselves in the form of attributes. HSF model .xml file has many elements. For instance, the system model is the first element and within the model is each asset element of the system. Also, within each asset element are the subsystem elements and constraint elements. The dependency elements reside within the subsystem elements. A subsystem element, its attributes, and the dependencies can be seen within Figure 5. Also, a constraint element with its relevant attributes can be seen in Figure 6. In order to improve the readability of the systemEval namespace and provide HSF with the ability to identify subsystems and constraints, subsystem names and constraint names were created. These names are attributes residing within the “SUBSYSTEM” element labeled as “SubsystemName” and within the “CONSTRAINT” element labeled as “constraintName”. Reading through each element and acquiring information within is a key feature of systemEval.

```
<SUBSYSTEM
      Type="SubsystemNode6"
      SubsystemName = "SubsystemNode6">
      <IC type="Double" key="Requirement6" value="0.0">
      </IC>
      <DEPENDENCY subsystemName="SubsystemNode1">
      </DEPENDENCY>
      <DEPENDENCY subsystemName="SubsystemNode2">
      </DEPENDENCY>
</SUBSYSTEM>
```

Figure 5: Subsystem .xml File

```
<CONSTRAINT
      value="0.12"
      subsystemName="SubsystemNode1"
      constraintName = "Constraint1"
      type="FAIL_IF_HIGHER">
      <STATEVAR type = "Double" key="Requirement1">
      </STATEVAR>
</CONSTRAINT>
```

Figure 6: Constraint .xml File

3.3 Creating HSF Log

Previously, HSF output log data via tab delimited .txt files only for created Schedules. With the implementation of the Log Data, Log, and Logger C# Classes of systemEval, the data regarding failed Schedules are recorded within HSF. After every failed Schedule, HSF has information regarding the failed subsystem or the violated constraint. Now, HSF can query the Logger class to create Log Data and store them in the Log. With the Log Data inside the Log, rates of occurrence of a subsystem failure or the rates of occurrence of a constraint violation can

be calculated. Other functions within the systemEval namespace input the Log to calculate those rates. Example representations of the Log Data and Log information are shown in Table 1 and Table 2. All the Log Data within the Log was specifically designed to capture the most important information regarding each failed Schedule.

3.4 Designing Log Data

In order to provide the user and HSF with as much information as possible, the Log Data class contained many properties regarding a failed Schedule. As seen in both Table 1 and Table 2, each column represents a property of the Log Data class. Some properties were necessary for rate calculations, but most were designed to provide the user and HSF enough information to analyze failed Schedules. In Table 2, instances of failed subsystems can be seen in the first two rows of the Log. The telltale sign of a failed subsystem is the 'null' value recorded for the constraint name. On the contrary, the last two rows of Table 2 represent instances of violated constraints. Since the constraint name does not contain a null value, the user and HSF know the Schedule failed due to a constraint violation. Whenever HSF specifies, the Log can be used to calculate rates of occurrence.

Table 1: Generated Log Data

Asset Name	Subsystem Name	Task Name	Target Name	Constraint Name	Violating State Variable	State Variable Value	Time Information
Asset1	Subsystem Node6	Task4	Ground Station1	null	null	0	25

Table 2: Generated Log

Asset Name	Subsystem Name	Task Name	Target Name	Constraint Name	Violating State Variable	State Variable Value	Time Information
Asset1	Subsystem Node6	Task4	Ground Station1	null	null	0	25
Asset1	Subsystem Node2	Task4	Ground Station3	null	null	0	17
Asset2	Subsystem Node4	Task2	Ground Station4	Constraint7	Requirement 4	0.84	86
Asset1	Subsystem Node1	Task5	Ground Station5	Constraint1	Requirement 1	0.73	56
Asset1	Subsystem Node9	Task2	Ground Station2	Constraint6	Requirement 8	0.6	27
Asset3	Subsystem Node11	Task5	Ground Station3	null	null	0	96
Asset1	Subsystem Node8	Task2	Ground Station5	Constraint5	Requirement 8	0.82	66
Asset1	Subsystem Node1	Task5	Ground Station1	Constraint1	Requirement 1	0.77	36
Asset1	Subsystem Node6	Task3	Ground Station4	Constraint4	Requirement 6	0.94	6
Asset2	Subsystem Node7	Task5	Ground Station4	Constraint9	Requirement 7	0.93	76

3.5 Log Analytics

Using all the organized data from the Log, systemEval tool can calculate subsystems that are most likely to fail and constraints that are most likely to be violated. The functionality of calculating rates is structured within the LogAnalyzer class of the systemEval namespace. The exact method that calculates the rates is the LogAnalyzer.Analyze method. With the information yielded by the LogAnalyzer class, the user can see specific issues within their system. Using that information, the user may be able to make deductions regarding their design. The method that outputs the rates information to the user is the LogAnalyzer.Post Process method. The user's deductions could result in improving in the system through design changes or just a better

understanding of the system. Not only can the rates be utilized by the user, but HSF's Scheduler can use the rates as well.

Using the information from LogAnalyzer.Analyze, HSF's Scheduler can improve run-time. Knowing the most likely to fail subsystems and the most likely to violate constraints, the Scheduler can check those parameters first by using the LogAnalyzer's reordering functionality. Those functions are within the LogAnalyzer.reorderCons and the LogAnalyzer.reorderSubs methods. Where, reorderCons creates the proper order of constraints checking and the reorderSubs produces the proper order of subsystems checking. The benefits of the LogAnalyzer class are ubiquitous throughout the architecture of the systemEval namespace.

3.6 Architecture & Logic

The functions described in the framework architecture had many reasons for their existence and method of execution. With many objectives, requirements, and constraints for the tool design, the logic of the systemEval namespace adhered to those ideas. Simplicity was a large focus of the tool to reduce run time and enhance readability. The overall flow diagram of the systemEval namespace can be seen in Figure 7. Each failed schedule has data extracted and stored in the Log Data class. Then the Logger class adds the Log Data to the Log class. After that, the LogAnalyzer class uses the current Log and analyzes it. If there has already been analytics perform on a previous log, the log analyzer class uses that information as well. The archeitecture of the systemEval namespace required defining many aspects and anticipated information from HSF.

Some of HSF's available information was already established before the development of the tool, but the Log Data, Logger, and Log classes had to be defined, designed, and used within

the systemEval namespace. Also, the .xml model input files were modified to include asset names and constraint names. Since, there was a need for evaluating constraint violations specifically. Therefore, constraints were assigned names to allow for ease of recognizing specific constraints. Before, constraints were only identified by their related subsystem identification numbers that they applied to. Thus, systemEval catered to using the previous .xml input files while slightly modifying them and anticipating new information provided by HSF. In addition, systemEval produced outputs similar to anticipated outputs by HSF.

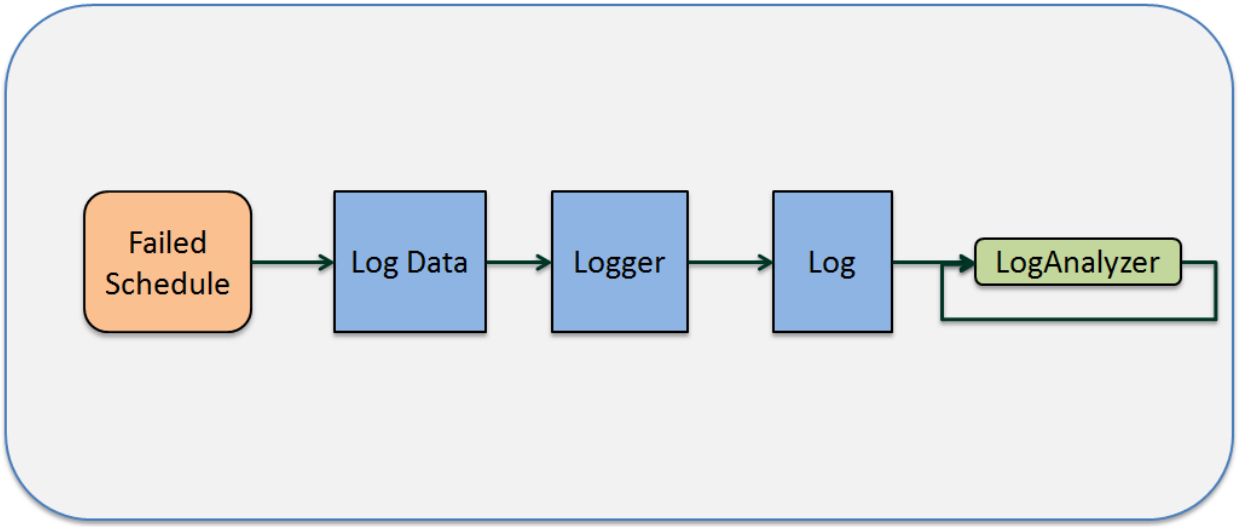

Figure 7: Flow Diagram

Since the dependencies and constraints were already written in the model .xml file, systemEval developed an .xml file parser to extract the relevant information. Within the C# LINQ library, exists many methods for .xml file parsing. The architecture utilized the LINQ library methods to iterate through the .xml file elements and retrieve information. Since the amount of assets, subsystems, state variables, and constraints are unknown, the .xml parser within systemEval recorded information in lists. Lists allow for information to be added to a collection of objects, whereas arrays do not. After reading all the subsystems and constraints into

lists, each list was stored in the correct C# class. Once the model .xml input is read-in, the systemEval namespace is capable of executing other features.

The systemEval requirements led to creating two modes types: Run Time mode and Post Process mode. The difference in mode types was to allow for HSF's fail-fast philosophy. Run Time mode would perform log analytics, but would not write any files. While the other mode, Post Process mode, would do the same, but would always write two comma delimited .csv files. Both were presented to the user to provide simplicity and the ability to further analyze a simulated system.

When in either mode, the user is able to independently analyze each subsystem and constraint. By attaining each failure rate of a subsystem or violating constraint, the user is then able to understand which subsystems or constraints are the least robust or limiting during the simulation. From there the user can modify their model or simulation to improve the system performances. Providing this ability to the user was seen as an improvement to HSF overall, which was one of the driving factors of creating the LogAnalyzer class. Another capability required in the LogAnalyzer Class was creating proper orders when checking subsystems and constraint violations.

The function of creating proper orders was seen as an improvement to HSF because it supplies to ability to improve HSF's fail-fast philosophy by reducing run-time. When proper orders are created, HSF's Scheduler is supplied with the ability to change the checking orders to reduce run-time. Producing the proper checking orders is a simple method that has the possibility to greatly improve HSF run-time. The proper order output is one of many helpful functions of systemEval.

Each function within systemEval had multiple reasons for their existence. Many had unique reasons, other functions shared similar reasons. Most of the reasoning was in favor of simplicity, stated requirements, functionality, or consistency. Each reason was seen to improve HSF framework as a whole by either benefitting the user or the HSF Scheduler. Although there are many capabilities within systemEval, they are presented in a simple and readable way for any user.

4. Test Cases

4.1 Procedure

The order of execution within systemEval is in a linear format with multiple options for user selection. First, the user is prompted with an action of selecting the model .xml input to be loaded into the systemEval namespace. After that, the tool reads through the .xml file and finds “SUBSYSTEM” and “CONSTRAINT” elements. Within each “SUBSYSTEM” element, each “SubsystemName” element and each “key” is found. Whereas in the “SUBSYSTEM” element, “SubsystemName” represents a subsystem within an asset and “key” represents a state variable within the subsystem. Also, within the “CONSTRAINT” element, each “subsystemName” element, “constraintName” element, “key” element and “value” element is found. Whereas in the “CONSTRAINT” element, “subsystemName” identifies a subsystem, “constraintName” identifies the constraint, “key” identifies the state variables constrained, and “value” identifies the subsystem’s constraint value. While reading for each relevant element, the subsystem information and constraint information are stored separately in specific C# classes. This functionality is already within HSF. However, during the development of the systemEval tool, the ability to read-in .xml files was not available. Therefore, systemEval created its own .xml reader similar to HSF’s. The subsystem class and constraint classes closely represented the classes within HSF.

Both the subsystems and the constraints are stored as their own C# class. For each subsystem exists a SubsystemClass class, which includes the subsystem name, the asset name were the subsystem resides in, and a list of the state variable names that reside within the subsystem. The subsystem name and asset name are stored as strings. Also, the state variable list

is a list of strings. In the Constraint class, reside the constraint name, the constraint value, and the list of state variables constrained by the constraint. The constraint name is stored as a string. Also, the value of each constraint is stored as a double. The constrained state variables are stored as a list of strings. The SubsystemClass class and Constraint class were created not only to be similar to HSF classes, but also to allow utilization of C# methods when locating specific data. In result, their existence not only allowed more C# methods to be used, but also aided to the readability of the systemEval namespace. Both classes provided assistance in the organization of information used within systemEval.

Using the SubsystemClass and Constraint classes, a state variable class can be created. Within the state variable class, StateVar, resides the name of the state variable, the name of the subsystem where the state variable is within, and the name of the constraints that apply to the state variable. Both the state variable name and the subsystem name are stored as strings. Where, the constraint names are stored as a list of strings. With all information regarding the system stored and organized, systemEval can evaluate failed schedules.

Once the initial .xml model input is loaded into systemEval, the user is prompted to specify the number of failed schedules to generate log data and then select a mode type. The user specifications are purely to create test cases. There are two different modes: Run Time mode and Post Process mode. If the user selects Run Time mode, the program will call the LogAnalyzer.Analyze method to calculate rates more frequently. Otherwise, the program will execute in the Post Process mode which calculates rates less frequently. Also, Post Process mode writes the log analytical data to two comma delimited .csv files named “subsytemFailures.csv” and “constraintViolations.csv”. The comma delimited .csv file type was chosen due to its commonality and readability for the user. Since .csv files can be opened with Microsoft Excel,

most users will be able to read the log analytical data. Once the log data has been read-in, both modes can begin evaluating subsystems and constraints.

First, both modes call the LogAnalyzer.Analyze method. In order to execute, any previous log analytics, the log, the list of all constraints, the list of all system subsystems, and the list of all state variables must be input into the LogAnalyzer.Analyze method. When executing, the LogAnalyzer.Analyze method initializes the numerous LogAnalyzer properties. First, the asset names, task names, and target names of a failed subsystem are stored as lists of strings. Next, the subsystem name of the failed subsystem as a string. Then, the calculated subsystem failure rate is stored as a double. After that, the constraint name of the violated constraint is stored as a string. Next, the constraint violation rate and the total number of failed schedules are both stored as doubles. Then, the names of the violating state variables and their relative subsystems are stored as list of strings. Lastly, the rates of violating subsystems are stored as a list of doubles. These properties are initialized to create LogAnalyzer classes within the LogAnalyzer.Analyze method.

To calculate rates and identify failing subsystems and violating constraints, the LogAnalyzer.Analyze method iterated through each subsystem, constraint, and state variable. Not only, were the key elements of a system thoroughly iterated, but many C# methods were utilized as well. These methods included List.Find(), List.FindAll(), foreach loops, List.Count(), List.Clear(), and List.Add(). During iteration, every instance a subsystem violation or constraint violation is counted. Once all the log data elements within the log are checked, the failure rates of each subsystem and the violation rates of each constraint are populated by dividing the failure counter or violation by the number of failed schedules and adding any previous calculated rates. Once the failure rates of each subsystem and the violation rates of each constraint are calculated,

the proper order of subsystem failure or constraint violation checking can be evaluated. In both modes, after the LogAnalyzer.Analyze method is called, the LogAnalyzer.reorderCons and the LogAnalyzer.reorderSubs methods can be called.

Currently, HSF checks constraints in a linear order. Using the calculated violation rates of each constraint, the proper order of constraint violation checking can be determined. The method LogAnalyzer.reorderCons can reorder a list of constraints and produce the proper order of constraint checking. In order to execute, the LogAnalyzer.reorderCons method requires an input of log analytical data and a list of constraints. Within LogAnalyzer.reorderCons method, each constraint is iterated and their violation rates are gathered. Then, the list of constraints is reorganized by violation rate in descending order. Lastly, the output is a new list of constraints in the proper constraint violation checking order. Similarly to constraint violation checking order, subsystem failure checking order can be reorganized.

Some subsystems have multiple dependent subsystems whose failures are checked before properly checking other dependent subsystems first. Using the calculated failure rates of each subsystem, the proper order of subsystem failure checking can be determined. Similar to LogAnalyzer.reorderCons method, LogAnalyzer.reorderSubs method require an input of log analytical data, but differs for the second input. For LogAnalyzer.reorderSubs, the second input is a list of subsystems instead of a list of constraints. Next, the LogAnalyzer.reorderSubs method iterates through each subsystem and their failure rates are extracted. Then, the list of subsystems is reorganized in descending order by failure rate. Lastly, a list of subsystems in the proper subsystem failure checking order is outputted. In Run Time mode, new log data would be logged, but in Post Process mode, one more method is called.

When in Post Process mode, the log analytical data must be recorded for the user. After all log analytical data is produced, the LogAnalyzer.Post Process method is called. Input to the LogAnalyzer.Post Process method is only the log analytical data. Once input, the method reorganizes the failed subsystems in descending order by failure rate. Similarly, the method also reorganizes the violated constraints in descending order by violation rate. After that, the method creates the "subsystemFailures.csv" file and writes each failed subsystem, their respective failure rate, asset name, failed task names, and failed target names. Then, the "subsystemFailures.csv" file is closed. After that, the method creates the "constraintViolations.csv" and writes each violated constraint, their respective violation rates, asset names, violating subsystems, violation subsystem rates, and violation state variables. Lastly, the "constraintViolation.csv" file is closed. Then the user has a .csv file of the log analytical data. Once every failed schedule log data is iterated, logged, and analyzed, the tool exits.

4.2 Test Case One Subsystem Tree

In order to validate systemEval's robustness, multiple subsystem trees were tested for the Logger and LogAnalyzer. Test Case One's subsystem tree seen in Figure 5 was created. As can be seen, there are three assets with a total of twelve subsystems. Within each asset, reside dependent subsystems, but the first asset and the second asset both have independent subsystems. Also, the third asset is dependent upon the first and second asset through subsystem dependencies between the assets. However, the first asset and the second asset are independent from each other. With the defined subsystem tree, each subsystem's dependencies and constraints were defined in a model .xml input. In total four test cases of different subsystem trees were implemented. However, the subsystem tree in Figure 5 was the most complex for reordering subsystem and constraint checking.

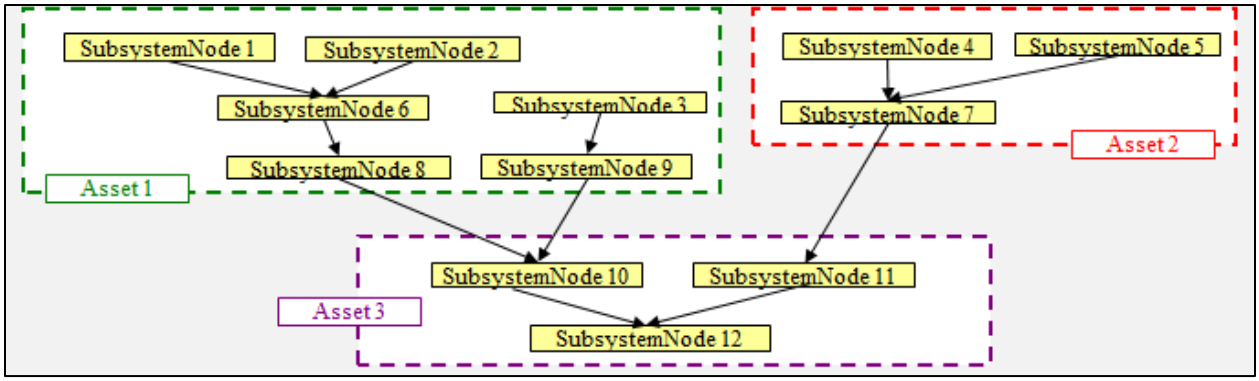

Figure 8: Subsystem Tree

4.3 Logger Results

All the log data was generated using with a GenRandLog class for every test case. Using the C# Random() method, the GenRandLog class created pseudorandom log data. The generated log data was sent to the Logger which added it to the Log. Ten generated log data of failed schedules were input to the Logger. An example of ten failed schedules' log data generated by GenRandLog class can be seen in Table 2. Both subsystem failures and constraint violations were randomly created in order for the LogAnalyzer to have mock data to process.

4.4 Test Case One Log Analyzer Results

Using the subsystem tree in Figure 8 and the log of ten schedules from Table 2, LogAnalyzer created results for subsystem failures and constraint violations. When in Post Process mode, two comma delimited.csv files of the results were outputted. The file contents can be seen in Table 3 and Table 4. Three other test cases with different subsystem trees were inputted for over 10,000+ failed schedules. To easily identify functionality, small scale results were included in this section.

Table 3: Test Case One Subsystem Failures

Failed Subsystem	Subsystem Failure Rate	Asset Name	Task Names	Target Names
SubsystemNode2	0.1	Asset1	Task4	GroundStation3
SubsystemNode8	0.1	Asset1	Task2	GroundStation1
SubsystemNode11	0.1	Asset3	Task5	GroundStation3
SubsystemNode1	0	Asset1		
SubsystemNode3	0	Asset1		
SubsystemNode6	0	Asset1		
SubsystemNode9	0	Asset1		
SubsystemNode4	0	Asset2		
SubsystemNode5	0	Asset2		
SubsystemNode7	0	Asset2		
SubsystemNode10	0	Asset3		
SubsystemNode12	0	Asset3		

Table 4: Test Case One Constraint Violations

Violated Constraint	Constraint Violation Rate	Asset Names	Violating Subsystems	Violating Subsystem Rates	Violating State Variables
Constraint1	0.1	Asset1	SubsystemNode1	0.1	Requirement1
Constraint4	0.1	Asset1	SubsystemNode6	0.1	Requirement6
Constraint5	0.1	Asset1	SubsystemNode8	0.1	Requirement8
Constraint6	0.1	Asset1	SubsystemNode9	0.1	Requirement9
Constraint7	0.1	Asset2	SubsystemNode4	0.1	Requirement4
Constraint9	0.1	Asset2	SubsystemNode7	0.1	Requirement7
Constraint10	0.1	Asset3	SubsystemNode10	0.1	Requirement10
Constraint2	0				
Constraint3	0				
Constraint8	0				
Constraint11	0				
Constraint12	0				

4.5 Test Case One Reordering Results

Using the same test case and log data, the reordering processes were tested. For subsystem checking, SubsystemNode1 and SubsystemNode2 were input. The output of the LogAnalyzer.reorderSubs method was SubsystemNode2 and SubsystemNode1. For constraint

checking, Constraint1 and Constraint2 were input. The output of the LogAnalyzer.reorderCons method was Constraint1 and Constraint2. Both methods output the proper classes in lists.

4.6 Test Case Two Log Analyzer Results

For this test case new log data was generated and analyzed for a different system. The new system only had six subsystems with two constraints within two assets. Having multiple state variables in constraints tested LogAnalyzer's robustness. The results of the subsystem failures can be seen in Table 5. Also, the results of the constraint violations can be seen in Table 6. The outputted results provided the ability to reorder the subsystem and constraint checking.

Table 5: Test Case Two Subsystem Failures

Failed Subsystem	Subsystem Failure Rate	Asset Name	Task Names	Target Names
SubsystemNode1	0.1	Asset1	Task4	GroundStation3
SubsystemNode4	0.1	Asset2	Task2	GroundStation1
SubsystemNode6	0.1	Asset2	Task5	GroundStation3
SubsystemNode2	0	Asset1		
SubsystemNode3	0	Asset1		
SubsystemNode5	0	Asset2		

Table 6: Test Case Two Constraint Violations

Violated Constraint	Constraint Violation Rate	Asset Names	Violating Subsystems	Violating Subsystem Rates	Violating State Variables
Constraint1	0.4	Asset1	SubsystemNode1	0.1	Requirement1
		Asset1	SubsystemNode2	0.1	Requirement2
		Asset1	SubsystemNode3	0.2	Requirement3
Constraint2	0.3	Asset2	SubsystemNode4	0.2	Requirement4
		Asset2	SubsystemNode5	0.1	Requirement5

4.7 Test Case Two Reordering Results

For Test Case Two, the reordering processes were tested using the results in Table 5 and Table 6. For subsystem checking, SubsystemNode1 and SubsystemNode2 were input. The output of the LogAnalyzer.reorderSubs method was SubsystemNode1 and SubsystemNode2. For constraint checking, Constraint1 and Constraint2 were input. The output of the LogAnalyzer.reorderCons method was Constraint1 and Constraint2. Again, both methods output the proper classes in lists.

4.8 Test Case Three (Aeolus) Log Analyzer Results

For Test Case Three, Aeolus' system had new log data generated and analyzed. Aeolus only had six subsystems with two constraints within one asset. Having a single asset was a very basic case, but Aeolus is a successful example in HSF. Also, Aeolus had state variables that were unconstrained. Therefore, testing LogAnalyzer on Aeolus once again further proved LogAnalyzer's ability. The results of the subsystem failures can be seen in Table 7. Also, the results of the constraint violations can be seen in Table 8. The outputted results provided the ability to reorder the subsystem and constraint checking.

Table 7: Test Case Three Subsystem Failures

Failed Subsystem	Subsystem Failure Rate	Asset Name	Task Names	Target Names
SubsystemNode1	0.1	Asset1	Task4	GroundStation3
SubsystemNode4	0.1	Asset1	Task2	GroundStation1
SubsystemNode6	0.1	Asset1	Task5	GroundStation3
SubsystemNode2	0	Asset1		
SubsystemNode3	0	Asset1		
SubsystemNode5	0	Asset1		

Table 8: Test Case Three Constraint Violations

Violated Constraint	Constraint Violation Rate	Asset Names	Violating Subsystems	Violating Subsystem Rates	Violating State Variables
Constraint1	0.4	Asset1	SubsystemNode6	0.4	Requirement6
Constraint2	0.3	Asset1	SubsystemNode4	0.3	Requirement4

4.9 Test Case Three (Aeolus) Reordering Results

For Test Case Three, the reordering processes were tested using the results in Table 5 and Table 6. For subsystem checking, SubsystemNode5 and SubsystemNode6 were input. In Aeolus, each subsystem has single dependencies. Therefore, the subsystem reordering tool would not need to be used. For practicality sake, the subsystem reordering tool was tested. The output of the LogAnalyzer.reorderSubs method was SubsystemNode6 and SubsystemNode5. For constraint checking, Constraint1 and Constraint2 were input. Constraint checking could still be reordered in Aeolus' system. The output of the LogAnalyzer.reorderCons method was Constraint1 and Constraint2. Again, both methods output the proper classes in lists.

4.10 Test Case Four Log Analyzer Results

For Test Case Four, another new system had new log data generated and analyzed. The new system only had four subsystems with two constraints within two assets. With multiple state variables within subsystems and constraints applying to multiple subsystems, testing LogAnalyzer on this system solidified LogAnalyzer's functionality. The results of the subsystem failures can be seen in Table 9. Also, the results of the constraint violations can be seen in Table 10. The outputted results provided the ability to reorder the subsystem and constraint checking.

Table 9: Test Case Four Subsystem Failures

Failed Subsystem	Subsystem Failure Rate	Asset Name	Task Names	Target Names
SubsystemNode1	0.1	Asset1	Task4	GroundStation3
SubsystemNode3	0.1	Asset2	Task2	GroundStation1
SubsystemNode4	0.1	Asset2	Task5	GroundStation3
SubsystemNode2	0	Asset1		

Table 10: Test Case Four Constraint Violations

Violated Constraint	Constraint Violation Rate	Asset Names	Violating Subsystems	Violating Subsystem Rates	Violating State Variables
Constraint1	0.4	Asset1	SubsystemNode1	0.1	Requirement1
		Asset1	SubsystemNode2	0.1	Requirement3
		Asset2	SubsystemNode3	0.1	Requirement5
		Asset2	SubsystemNode4	0.1	Requirement7
Constraint2	0.3	Asset1	SubsystemNode1	0.1	Requirement2
		Asset1	SubsystemNode2	0.1	Requirement4
		Asset2	SubsystemNode3	0.1	Requirement6

4.11 Test Case Four Reordering Results

In Test Case Four, the reordering functions were tested using the results in Table 9 and Table 10. For subsystem checking, SubsystemNode1 and SubsystemNode2 were input. The output of the LogAnalyzer.reorderSubs method was SubsystemNode1 and SubsystemNode2. For constraint checking, Constraint1 and Constraint2 were input. The output of the LogAnalyzer.reorderCons method was Constraint1 and Constraint2. The reordering functions output the proper lists comprised of the correct classes.

4.12 Test Case Five Log Analyzer Results

To validate LogAnalyzer's abilities, thirty failed schedules were tested using the same model in Test Case Four. More schedules created more differences with rates of occurrences. The thirty failed schedules were implemented to add more clarity to LogAnalyzer's capabilities. The results of the subsystem failures can be seen in Table 11. Also, the results of the constraint violations can be seen in Table 12. The outputted results once again, provided the ability to reorder the subsystem and constraint checking.

Table 11: Test Case Five Subsystem Failures

Failed Subsystem	Subsystem Failure Rate	Asset Name	Task Names	Target Names
SubsystemNode1	0.1	Asset1	Task4	GroundStation3
			Task3	GroundStation1
			Task1	GroundStation4
SubsystemNode4	0.1	Asset2	Task5	GroundStation3
			Task3	GroundStation2
			Task2	GroundStation4
SubsystemNode2	0.066666667	Asset1	Task1	GroundStation3
			Task4	GroundStation1
SubsystemNode3	0.033333333	Asset2	Task2	GroundStation1

Table 12: Test Case Five Constraint Violations

Violated Constraint	Constraint Violation Rate	Asset Names	Violating Subsystems	Violating Subsystem Rates	Violating State Variables
Constraint2	0.37	Asset2	SubsystemNode3	0.23	Requirement5
					Requirement6
		Asset2	SubsystemNode4	0.13	Requirement7
					Requirement8
Constraint1	0.33	Asset1	SubsystemNode1	0.13	Requirement1
					Requirement2
		Asset1	SubsystemNode2	0.2	Requirement3
					Requirement4

4.13 Test Case Five Reordering Results

For the thirty failed schedules, the reordering functions were tested using the results in Table 11 and Table 12. For subsystem checking, SubsystemNode2, SubsystemNode3, and SubsystemNode4 were input. The output of the LogAnalyzer.reorderSubs method was SubsystemNode4, SubsystemNode2, and SubsystemNode3. More subsystems were chosen to be input to further prove the reorderSubs robustness. For constraint checking, Constraint1 and Constraint2 were input. The output of the LogAnalyzer.reorderCons method was Constraint1 and Constraint2. Once again, the reordering functions output the proper lists comprised of the correct classes.

4.14 Analysis

For all functionalities, the results matched what would be expected as outputs. Also, the values of the outputs met expectations designed by the test cases. Each input of the test cases was designed to created predictable results for each mode. The designs of the inputs created predictions that matched the outputs from systemEval. By designing the inputs purposefully, functions of systemEval could easily be discerned.

4.15 Logger Analysis

All the data output by the Logger was exactly as expected. The input for the Logger was each log data generated. The Logger input the Log Data and then updated the Log every time the Logger was called. An example of the Logger successfully creating a Log can be seen in Table 1. Even after 10,000+ failed schedules the Logger was successfully adding to the Log. Since, the Logger output performed as expected, the Logger was verified to be properly functioning.

4.16 LogAnalyzer Analysis

For the LogAnalyzer, all the results met exactly what was expected from the inputs. The log data of subsystems were designed to show predictable failure rates and obvious instances of needing reordering. For instance, the Constraint 1 can be seen in Table 5 as violating 10% of all failed schedules. When looking at the log data for Constraint 1 in the test case shown in Table 1, it can be seen that Constraint 1 is violated once of all the instances of the failed schedules. Using simple mathematics, that would yield a violation rate of ten percent which is the exact violation rate seen in Table 5 of the LogAnalyzer output. In another instance, the failure rate calculations are working properly as seen in the failure rate of Subsystem 2. The Subsystem is seen to have failed once in the Log Data. Thus, yielding a ten percent failure rate as can be seen in Table 1 of the log data. All other subsystems' failure rates and constraint violations were correctly calculated in each test case. Not only did LogAnalyzer calculate rates, but output relevant information to the user.

Also, user information for each subsystem failure and constraint violation was outputted correctly. Output in .csv files, all relevant information was organized to the correct corresponding failing subsystems and violating constraints. Also, all the log data information was correct. As seen in Table 2, Constraint 1 constrains Requirement1 which resides in SubsystemNode1. That information can be verified within the .xml file seen in Appendix A. Outputting important information to the user was one of the functions of LogAnalyzer. Another function, reordering, performed just as well.

4.17 Reordering Analysis

As stated earlier in the results, the LogAnalyzer.reorderSub and the LogAnalyzer.reorderCon produced results as expected. As can be seen in Table 4, SubsystemNode2 fails more often than SubsystemNode1. Therefore, SubsystemNode2 should be checked first. Similarly, Constraint1 is violated more often than Constraint2. Thus, Constraint1 should be checked before Constraint2. The proper outputs of both reordering methods of the LogAnalyzer showed the reordering process is validated.

4.18 Test Cases Conclusion

By designing and executing complex test cases, the functions of systemEval were validated. The inputs were designed to be diverse in order to create different expected outputs and extensive cases. All the outputs in both modes were correct in format and accuracy. With the all functions of both modes performing properly, the systemEval namespace was validated as a whole. The complex test cases verified systemEval's robustness for any system output from HSF.

5. Future Work

The success of the systemEval namespace allows for numerous opportunities to modify and improve HSF as a whole. There are currently three main ideas that can be utilized. First, use the methods of systemEval to determine if independent constraints are checked efficiently. When HSF is validating constraints, it follows the subsystem tree for comparing subsystem log data against constraint values. By looking at the independent subsystems and their failure rates, constraint validation can be improved for processing run-time.

The idea of checking whether independent subsystems are more likely to fail requires defining a new organizational element of the subsystem tree. Discussions have created an idea of subsystem tree 'levels'. Where, a level includes all subsystems that are equal in processing order. Then each subsystem can be evaluated if processing in the right order. Simply, the failure rates would be calculated and compared to determine instances of poor processing. The LogAnalyzer class provides this capability, some implementation should be simple.

Steps to calculate failure rates of subsystem levels require deeply defining subsystem tree levels. Once the subsystem tree levels have been accurately defined, utilizing the methods of the systemEval namespace to perform constraint checks should be simple. The method would essentially emulate the dependencies checking method. Another area of work is in the realm of modifying the Scheduler during run-time.

Actively adapting the Scheduler for better processing during run-time should be an easy integration for subsystemEval. The idea would be to tell the Scheduler the new order of

constraints and subsystems. In result, processing time would be actively improved by implementing HSF fail-fast philosophy. The decision making feature would be the most interesting function. Deciding when to adapt the Scheduler could be improved upon with a robust machine learning tool. In order to provide an adequate decision making tool, thorough research should be conducted.

In general, machine learning and probability measurement methods were researched for the systemEval namespace. After many discussions with faculty members of Cal Poly including those with experience in statistics and artificial intelligence, unsupervised machine learning was determined to be a possible practical method for HSF. Then, following further discussions and investigations, creating a machine learning tool in addition to the features of the systemEval namespace was deemed to be out of scope for a Thesis for a Master of Science in Aerospace Engineering. Thus, this discussion has been created. Not only will research be described, but suggestions for future work using the systemEval namespace will be outlined.

In regards to simple probability measurements, a priori, a posteriori (also known as empirical), and Bayesian statistics were researched. The a priori method and Bayesian method, requires knowledge about a sample prior to gathering data. Thus, in the applications of the systemEval namespace, were impractical. Simulated systems are often vastly too different to manifest an accurate deduction before analyzing data. However, there are Bayesian methods that can make effective deductions. Bayesian statistics are a growing and highly favored method amongst most statisticians and those in the field of artificial intelligence.

Similar to a priori methods, Bayesian statistics make deductions prior to acquiring data. Then after acquiring the data, Bayesian statistics can make more deductions after analyzing the

data and using the original deduction. In practice, Bayesian statistics was certainly the most powerful way to acquire accurate probability methods. However, Bayesian statistical methods are usually computationally expensive and process time consuming with large amounts of data. In practicality of the systemEval namespace, Bayesian statistics was far too complex for the scope.

Lastly, a posteriori, or empirical probability measurement, is simply taking post processed data and measuring the rate of occurrence. Empirical probability was chosen to be the best method of measurement due to simplicity and the effectiveness within the systemEval namespace. The simplicity of empirical method was not only effective, but most likely a familiar method to anyone working within HSF. Although more accurate methods for probability measurement exist, a posteriori probability was the most practical. The other realm of research was machine learning.

There were many options for machine learning when determining the needs for a machine learning Scheduler within HSF. The three most notable methods were Gaussian, Markov decision processes, and Bayesian inference. All in all, Bayesian inference seems to be the most promising method. Each had their benefits individually. However, in many cases the methods could use each other.

The Markov decision process (MDP) requires computing an optimal value-function. With small numbers of states, MDP can be solved with linear programming. Conventionally, MDP is most effective using dynamic programming. One focus to investigate is the term curse-of-dimensionality which describes how the number of states grows exponentially as the number of state variables grow. Thus, large numbers of states suffer from the curse when implementing the

MDP. Some MDP solutions include using the approximate value function, instead of the exact value. These include Reduced Linear Programming (RLP) and Generalized Reduced Linear Programming (GRLP).[10] By linearizing the MDP, the exact value of optimal function is not needed to be calculated. Therefore, it may be possible to calculate large states using MDP without excessive computations.[11] One key benefit of MDP is producing confidence intervals to aid in decision making.[2] A possible improvement to MDP is utilizing forgetting methods which in result improve the ability to learn.[3] In order to fail quickly, systemEval did not use MDP due to its complexity and the possibility of large states. Similar to MDP, Bayesian statistics showed promise for effective machine learning methods.

Bayesian classifiers have gained popularity recently and have been performing quite well. Strong assumptions must be made when using Bayesian inference regarding data generation. The Naïve Bayesian classifier is a very simple model, where it assumes that all attributes are independent of each other. This is also known as the "naïve Bayes assumption." Therefore, gaining the "naïve" term in the method, however this rarely applies in real world examples. A great benefit of Bayesian inference is that approximations can be inaccurate while classification accuracy will be high. Two common event models that use the naïve Bayes assumption are multi-variate Bernoulli and multinomial. At small data variations, the multi-variate Bernoulli method performs better than the multinomial method. However, with large data differences the multinomial performs better than the multi-variate Bernoulli.[12] Bayesian inference has more options than naïve Bayes classification, but both seem worthy of further investigation.

Unsupervised Bayesian learning is a logical method to implement within HSF. Due to the uncertainty regarding the data, HSF should use a method that accounts for those situations. New methods are being created to utilize statistical clusters to process data.[1] These methods seem

logically powerful, but are new and still establishing their ability. Many research institutions are developing their own cluster methods. These methods have been seen including the implementation of Gaussian models, Metropolis-Hastings algorithms, and Markov chain Monte Carlo (MCMC).[5,6] Unsupervised learning seems to be the most logical method of Bayesian inference, but once again seems to be too computationally extensive for large datasets. However, at small data sets the methods are successful with small computations. One method that seems promising for Bayesian unsupervised learning is sparse unsupervised learning.

Typically, spike-and-slab sparse Bayesian methods perform well even on a computational budget. This method is desirable when there are many underlying factors that could explain the data. Also, the method is desirable when a subset can explain the data and the subset is different for each observation.[14] The spike-and-slab method seems the most promising Bayesian unsupervised learning method. Other considerations for unsupervised learning include taking advantage of hyperparameters which are parameters of other distributions.[16] Unsupervised learning can be used on a desktop computer with datasets of up to a few thousands. Being accurate and computationally efficient are needs of HSF which sparse Bayesian learning seems to be able to provide.

For decision making, one emphasis in order to ensure good results is acquiring adequate data. As this is not always the case, methods should be utilized to aid in making the best decisions possible. The idea is to structure thinking without ignoring important features of a problem.[8] Therefore, the methods used to aid in decision making should minimize uncertainty. When considering any machine learning method, the purpose of the tool should consistently be evaluated.

Most of Bayesian epistemology relies on an epistemic intuition. Otherwise, Bayesian epistemology draws much of its power from the functions of mathematical probability theory, manifesting a mathematical intuition.[17] In order for Bayesian inference to be successful, assumptions and knowledge about data should be accurate. The essence of Bayesian classification is making determinations about data before it is measured. When implementing Bayesian machine learning, the depth of Bayesian epistemology should be considered.

All methods regarding machine learning are computationally expensive. However, newer methods are being developed and verified to produce accurate results quickly at low computational cost. As a recommendation to improve HSF Scheduler, any future work should investigate Bayesian inference. The Bayesian machine learning methods are currently establishing themselves and possibly there is a method that suits HSF. Open-source Bayesian methods are another option that should be investigated. There possibly could be open-source methods that only should be slightly modified to meet HSF's needs. Pursuing a Bayesian inference machine learning tool could greatly improve the efficiency of HSF as a whole.

All discussed methods should be furthered investigated. However, there are some methods that are more promising than others. For instance, Bayesian inference could be very useful, but is still proving its ability to be computationally efficient. All methods seem to be effective, but many are more computationally expensive than what is worth to HSF. When pursuing a machine learning algorithm, greatly research the available options and weigh the computational cost against the processing improvement. Also, consider open-source machine learning software. It is possible with good knowledge of machine learning; open source tools could be modified to meet the needs of HSF Scheduler. A machine learning tool would greatly improve HSF and the method of choice should be properly determined with thorough research.

Finally, an area for possible improvement is investigating reasons why Schedules were not created. An idea for doing so is creating a sensitivity analysis on Schedule constraints. By pursuing an analysis, constraints and log data can be compared to identify instances when log data barely violated constraints. These constraints would be deemed as sensitive and the user could possibly adjust their system to accommodate them. Being able to determine sensitive constraints could provide the user with more understanding of their system as whole.

6. Conclusion

HSF is a powerful tool with many capabilities in a growing field. Utilizing the powers of model-based engineering, HSF can provide users with quick results with low cost. However, HSF has been limited in providing users with subsystem failures and constraint violations information. One of the main reasons of the development of systemEval was to fill that void. Now, HSF has a tool that can provide the data of each failed Schedule and the rates of subsystem failures and constraint violations. Providing failed Schedule information to the user was one of the improvements to HSF through systemEval.

One of HSF's main philosophy is to fail-fast in order to provide the user with quick results. Systems with multiple constraints and subsystems with multiple dependencies could create an inefficient processing order. Thus, there is a possibility to decrease run-time by rearranging that order. With systemEval, HSF's Scheduler can reorder the subsystem and constraint checking to adapt and reduce inefficient processes. The information of better processes, could greatly improve HSF run-time.

Each improvement to HSF in systemEval was designed and tested for complex cases. Two modes were created to provide the user with more ability. These modes and their respective functions were verified in complex test cases. Thus, the systemEval namespace was validated as a whole. The successful development of systemEval will improve HSF to being a more informative framework that runs more effectively.

REFERENCES

1. Agius, Pahaedra, Yiming Ying, and Colin Campbell. "Bayesian Unsupervised Learning with Multiple Data Types." *ResearchGate*. Department of Engineering Mathematics, University of Bristol, n.d. Web. 15 Dec. 2015.

2. Chang, Hyeong Soo, Michael C. Fu, Jiaqiao Hu, and Steven I. Marcus. "Simulation-Based Algorithms for Markov Decision Processes | Hyeong Soo Chang | Springer." *Simulation-Based Algorithms for Markov Decision Processes | Hyeong Soo Chang | Springer*. Springer, 6 Oct. 2006. Web. 15 Dec. 2015.

3. Chen, Jun, Chaokun Wang, and Jianmin Wang. "A Personalized Interest-Forgetting Markov Model for Recommendations." *ResearchGate*. Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, n.d. Web. 15 Dec. 2015.

4. Estefan, Jeffrey A. "Survey of Model-Based Systems Engineering (MBSE) Methodologies." *INCOSE MBSE Initiative* (2008): 1-70. California Institute of Technology, 23 May 2008. Web. 2016.

5. Gaiffas, Stephane, and Bertrand Michel. "Sparse Bayesian Unsupervised Learning." *ResearchGate*. Universite Pierre Et Marie Curie, 3 Feb. 2014. Web. 15 Dec. 2015.

6. Gilks, W.R., S. Richardson, and D.J. Spiegelhalter. "Markov Chain Monte Carlo in Practice." Chapman & Hall, 1996. Web. 15 Dec. 2015.

7. Hart, Laura E. "Introduction To Model-Based System Engineering (MBSE) and SysML." *Http://www.incose.org/docs/default-source/delaware-valley/mbse-overview-incose-30-july-2015.pdf?sfvrsn=0*. Lockheed Martin, 30 July 2015. Web. 30 Mar. 2016.

8. Holsinger, Kent E. "Decision Making Under Uncertainty: Statistical Decision Theory." *Stanford University Creative Commons* (2013): n. pag. *Creativecommons.org*. 2013. Web. 15 Dec. 2015.

9. Kirkpatrick, Brian. "“PICASSO” INTERFACE FOR HORIZON SIMULATION FRAMEWORK." *Www.calpoly.edu*. California Polytechnic State University, Aug. 2010. Web. 15 Dec. 2015.

10. Lakshminarayanan, Chandrashekar, and Shalabh Bhatnagar. "A Generalized Reduced Linear Program for Markov Decision Processes." *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence* (n.d.): n. pag. Print.

11. Li, Meilun, Zhikun She, Andrea Turrini, and Lijun Zhang. "Preference Planning for Markov Decision Processes." *Aaai.org*. Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, n.d. Web. 15 Dec. 2015.

12. London, Brian Nathaniel. "A Model-Based Systems Engineering Framework for Concept Development." (2012): 1-152. Massachusetts Institute of Technology, 2012. Web. 2016.

13. McCallum, Andrew, and Kamal Nigam. "A Comparison of Event Models for Naive Bayes Text Classification." *A Comparison of Event Models for Naive Bayes Text Classification*. Just Research, n.d. Web. 15 Dec. 2015.

14. Mohamed, Shakir, Katherine A. Heller, and Zoubin Ghahramani. "Bayesian and L1 Approaches to Sparse Unsupervised Learning." *ResearchGate*. Proceedings of the 29th International Conference on Machine Learning, 2012. Web. 15 Dec. 2015.

15. O'Connor, Cory M. "Horizon: A System Modeling and Simulation Framework for Systems Engineering Utility Analysis." California Polytechnic State University, June 2007. Web. 15 Dec. 2015.

16. Rasmussen, Carl Edward. "Gaussian Processes in Machine Learning." *Tuebingin.mpg.de/~carl*. Max Planck Institute for Biological Cybernetics, n.d. Web. 15 Dec. 2015.

17. Sprenger, Stephan Hartmann And Jan, and Jan Sprenger. "BAYESIAN EPISTEMOLOGY." *BAYESIAN EPISTEMOLOGY* (2010): 1-19. *Stephanhartmann.org*. University of Gieben, 17 Aug. 2010. Web. 15 Dec. 2015.

Appendix A – Test Case One Model .xml File

```
<MODEL>

      <ASSET
            AssetName="Asset1">
            <SUBSYSTEM
                  Type="SubsystemNode1"
                  SubsystemName = "SubsystemNode1">
                  <IC type="Double" key="Requirement1" value="0.0"></IC>
            </SUBSYSTEM>
            <SUBSYSTEM
                  Type="SubsystemNode2"
                  SubsystemName = "SubsystemNode2">
                  <IC type="Double" key="Requirement2" value="0.0"></IC>
            </SUBSYSTEM>
            <SUBSYSTEM
                  Type="SubsystemNode3"
                  SubsystemName = "SubsystemNode3">
                  <IC type="Double" key="Requirement3" value="0.0"></IC>
            </SUBSYSTEM>
            <SUBSYSTEM
                  Type="SubsystemNode6"
                  SubsystemName = "SubsystemNode6">
                  <IC type="Double" key="Requirement6" value="0.0"></IC>
                  <DEPENDENCY
subsystemName="SubsystemNode1"></DEPENDENCY>
                  <DEPENDENCY
subsystemName="SubsystemNode2"></DEPENDENCY>
            </SUBSYSTEM>
            <SUBSYSTEM
                  Type="SubsystemNode8"
                  SubsystemName = "SubsystemNode8">
                  <IC type="Double" key="Requirement8" value="0.0"></IC>
                  <DEPENDENCY
subsystemName="SubsystemNode6"></DEPENDENCY>
            </SUBSYSTEM>
            <SUBSYSTEM
                  Type="SubsystemNode9"
                  SubsystemName = "SubsystemNode9">
                  <IC type="Double" key="Requirement9" value="0.0"></IC>
                  <DEPENDENCY
subsystemName="SubsystemNode3"></DEPENDENCY>
            </SUBSYSTEM>
      </ASSET>

      <ASSET
```

```
            AssetName="Asset2">
            <SUBSYSTEM
                  Type="SubsystemNode4"
                  SubsystemName = "SubsystemNode4">
                  <IC type="Double" key="Requirement4" value="0.0"></IC>
            </SUBSYSTEM>
            <SUBSYSTEM
                  Type="SubsystemNode5"
                  SubsystemName = "SubsystemNode5">
                  <IC type="Double" key="Requirement5" value="0.0"></IC>
            </SUBSYSTEM>
            <SUBSYSTEM
                  Type="SubsystemNode7"
                  SubsystemName = "SubsystemNode7">
                  <IC type="Double" key="Requirement7" value="0.0"></IC>
                  <DEPENDENCY
subsystemName="SubsystemNode4"></DEPENDENCY>
                  <DEPENDENCY
subsystemName="SubsystemNode5"></DEPENDENCY>
            </SUBSYSTEM>
      </ASSET>

      <ASSET
            AssetName="Asset3">
            <SUBSYSTEM
                  Type="SubsystemNode10"
                  SubsystemName="SubsystemNode10">
                  <IC type="Double" key="Requirement10"
value="0.0"></IC>
                  <DEPENDENCY
subsystemName="SubsystemNode8"></DEPENDENCY>
                  <DEPENDENCY
subsystemName="SubsystemNode9"></DEPENDENCY>
            </SUBSYSTEM>
            <SUBSYSTEM
                  Type="SubsystemNode11"
                  SubsystemName="SubsystemNode11">
                  <IC type="Double" key="Requirement11"
value="0.0"></IC>
                  <DEPENDENCY
subsystemName="SubsystemNode7"></DEPENDENCY>
            </SUBSYSTEM>
            <SUBSYSTEM
                  Type="SubsystemNode12"
                  SubsystemName = "SubsystemNode12">
                  <IC type="Double" key="Requirement12"
value="0.0"></IC>
                  <DEPENDENCY
subsystemName="SubsystemNode10"></DEPENDENCY>
                  <DEPENDENCY
subsystemName="SubsystemNode11"></DEPENDENCY>
            </SUBSYSTEM>
```

```
      </ASSET>

<CONSTRAINT
                  value="0.12"
                  constraintName = "Constraint1"
                  type="FAIL_IF_HIGHER">
                  <STATEVAR type = "Double"
key="Requirement1"></STATEVAR>
            </CONSTRAINT>
            <CONSTRAINT
                  value="0.19"
                  constraintName = "Constraint2"
                  type="FAIL_IF_HIGHER">
                  <STATEVAR type = "Double"
key="Requirement2"></STATEVAR>
            </CONSTRAINT>
            <CONSTRAINT
                  value="0.64"
                  constraintName = "Constraint3"
                  type="FAIL_IF_HIGHER">
                  <STATEVAR type = "Double"
key="Requirement3"></STATEVAR>
            </CONSTRAINT>
            <CONSTRAINT
                  value="0.13"
                  constraintName = "Constraint4"
                  type="FAIL_IF_HIGHER">
                  <STATEVAR type = "Double"
key="Requirement6"></STATEVAR>
            </CONSTRAINT>
            <CONSTRAINT
                  value="0.82"
                  constraintName = "Constraint5"
                  type="FAIL_IF_HIGHER">
                  <STATEVAR type = "Double"
key="Requirement8"></STATEVAR>
            </CONSTRAINT>
            <CONSTRAINT
                  value="0.54"
                  constraintName = "Constraint6"
                  type="FAIL_IF_HIGHER">
                  <STATEVAR type = "Double"
key="Requirement9"></STATEVAR>
            </CONSTRAINT>
<CONSTRAINT
                  value="0.79"
                  constraintName = "Constraint7"
                  type="FAIL_IF_HIGHER">
                  <STATEVAR type = "Double"
key="Requirement4"></STATEVAR>
            </CONSTRAINT>
            <CONSTRAINT
```

```
                value="0.24"
                constraintName = "Constraint8"
                type="FAIL_IF_HIGHER">
                <STATEVAR type = "Double"
key="Requirement5"></STATEVAR>
          </CONSTRAINT>
          <CONSTRAINT
                value="0.89"
                constraintName = "Constraint9"
                type="FAIL_IF_HIGHER">
                <STATEVAR type = "Double"
key="Requirement7"></STATEVAR>
          </CONSTRAINT>
<CONSTRAINT
                value="0.10"
                constraintName = "Constraint10"
                type="FAIL_IF_HIGHER">
                <STATEVAR type = "Double"
key="Requirement10"></STATEVAR>
          </CONSTRAINT>
          <CONSTRAINT
                value="0.49"
                constraintName = "Constraint11"
                type="FAIL_IF_HIGHER">
                <STATEVAR type = "Double"
key="Requirement11"></STATEVAR>
          </CONSTRAINT>
          <CONSTRAINT
                value="0.41"
                constraintName = "Constraint12"
                type="FAIL_IF_HIGHER">
                <STATEVAR type = "Double"
key="Requirement12"></STATEVAR>
          </CONSTRAINT>

</MODEL>
```

Appendix B – Test Case Two Model .xml File

```
<MODEL>

      <ASSET
            AssetName="Asset1">
            <SUBSYSTEM
                  Type="SubsystemNode1"
                  SubsystemName = "SubsystemNode1">
                  <IC type="Double" key="Requirement1" value="0.0"></IC>
            </SUBSYSTEM>
            <SUBSYSTEM
                  Type="SubsystemNode2"
                  SubsystemName = "SubsystemNode2">
                  <IC type="Double" key="Requirement2" value="0.0"></IC>
            </SUBSYSTEM>
            <SUBSYSTEM
                  Type="SubsystemNode3"
                  SubsystemName = "SubsystemNode3">
                  <IC type="Double" key="Requirement3" value="0.0"></IC>
            </SUBSYSTEM>
      </ASSET>

      <ASSET
            AssetName="Asset2">
            <SUBSYSTEM
                  Type="SubsystemNode4"
                  SubsystemName = "SubsystemNode4">
                  <IC type="Double" key="Requirement4" value="0.0"></IC>
            </SUBSYSTEM>
            <SUBSYSTEM
                  Type="SubsystemNode5"
                  SubsystemName = "SubsystemNode5">
                  <IC type="Double" key="Requirement5" value="0.0"></IC>
            </SUBSYSTEM>
            <SUBSYSTEM
                  Type="SubsystemNode6"
                  SubsystemName = "SubsystemNode6">
                  <IC type="Double" key="Requirement6" value="0.0"></IC>
                  <DEPENDENCY
subsystemName="SubsystemNode4"></DEPENDENCY>
                  <DEPENDENCY
subsystemName="SubsystemNode5"></DEPENDENCY>
            </SUBSYSTEM>
      </ASSET>

<CONSTRAINT
                  value="0.12"
                  constraintName = "Constraint1"
                  type="FAIL_IF_HIGHER">
```

```
                <STATEVAR type = "Double"
key="Requirement1"></STATEVAR>
                <STATEVAR type = "Double"
key="Requirement2"></STATEVAR>
                <STATEVAR type = "Double"
key="Requirement3"></STATEVAR>
           </CONSTRAINT>
<CONSTRAINT
                value="0.79"
                constraintName = "Constraint2"
                type="FAIL_IF_HIGHER">
                <STATEVAR type = "Double" key="Requirement4"
></STATEVAR>
                <STATEVAR type = "Double"
key="Requirement5"></STATEVAR>
                <STATEVAR type = "Double"
key="Requirement6"></STATEVAR>
           </CONSTRAINT>

</MODEL>
```

Appendix C – Test Case Three Aeolus Model .xml File

```
<MODEL>

      <ASSET
            AssetName="Asset1">
            <SUBSYSTEM
                  Type="SubsystemNode1"
                  SubsystemName = "SubsystemNode1">
                  <IC type="Double" key="Requirement1" value="0.0"></IC>
            </SUBSYSTEM>
            <SUBSYSTEM
                  Type="SubsystemNode2"
                  SubsystemName = "SubsystemNode2">
                  <IC type="Double" key="Requirement2" value="0.0"></IC>
<DEPENDENCY subsystemName="SubsystemNode1"></DEPENDENCY>
            </SUBSYSTEM>
            <SUBSYSTEM
                  Type="SubsystemNode3"
                  SubsystemName = "SubsystemNode3">
                  <IC type="Double" key="Requirement3" value="0.0"></IC>
<DEPENDENCY subsystemName="SubsystemNode2"></DEPENDENCY>
            </SUBSYSTEM>
            <SUBSYSTEM
                  Type="SubsystemNode4"
                  SubsystemName = "SubsystemNode4">
                  <IC type="Double" key="Requirement4" value="0.0"></IC>
<DEPENDENCY subsystemName="SubsystemNode3"></DEPENDENCY>
            </SUBSYSTEM>
<SUBSYSTEM
                  Type="SubsystemNode5"
                  SubsystemName = "SubsystemNode5">
                  <IC type="Double" key="Requirement5" value="0.0"></IC>
<DEPENDENCY subsystemName="SubsystemNode4"></DEPENDENCY>
            </SUBSYSTEM>
            <SUBSYSTEM
                  Type="SubsystemNode6"
                  SubsystemName = "SubsystemNode6">
                  <IC type="Double" key="Requirement6" value="0.0"></IC>
<DEPENDENCY subsystemName="SubsystemNode5"></DEPENDENCY>
            </SUBSYSTEM>
      </ASSET>

            <CONSTRAINT
                  value="0.25"
                  subsystemName="SubsystemNode6"
                  constraintName = "Constraint1"
                  type="FAIL_IF_HIGHER">
                  <STATEVAR type = "Double"
key="Requirement6"></STATEVAR>
            </CONSTRAINT>
```

```
            <CONSTRAINT
                  value="0.7"
                  subsystemName="SubsystemNode4"
                  constraintName = "Constraint2"
                  type="FAIL_IF_HIGHER">
                  <STATEVAR type = "Double"
key="Requirement4"></STATEVAR>
            </CONSTRAINT>


</MODEL>
```

Appendix D – Test Case Four Model .xml File

```
<MODEL>

      <ASSET
            AssetName="Asset1">
            <SUBSYSTEM
                  Type="SubsystemNode1"
                  SubsystemName = "SubsystemNode1">
                  <IC type="Double" key="Requirement1" value="0.0"></IC>
                  <IC type="Double" key="Requirement2" value="0.0"></IC>
            </SUBSYSTEM>
            <SUBSYSTEM
                  Type="SubsystemNode2"
                  SubsystemName = "SubsystemNode2">
                  <IC type="Double" key="Requirement3" value="0.0"></IC>
                  <IC type="Double" key="Requirement4" value="0.0"></IC>
                  <DEPENDENCY
subsystemName="SubsystemNode1"></DEPENDENCY>
            </SUBSYSTEM>
      </ASSET>

      <ASSET
            AssetName="Asset2">
            <SUBSYSTEM
                  Type="SubsystemNode3"
                  SubsystemName = "SubsystemNode3">
                  <IC type="Double" key="Requirement5" value="0.0"></IC>
                  <IC type="Double" key="Requirement6" value="0.0"></IC>
            </SUBSYSTEM>
            <SUBSYSTEM
                  Type="SubsystemNode4"
                  SubsystemName = "SubsystemNode4">
                  <IC type="Double" key="Requirement7" value="0.0"></IC>
                  <IC type="Double" key="Requirement8" value="0.0"></IC>
                  <DEPENDENCY
subsystemName="SubsystemNode3"></DEPENDENCY>
            </SUBSYSTEM>
      </ASSET>

      <CONSTRAINT
            value="0.12"
            constraintName = "Constraint1"
            type="FAIL_IF_HIGHER">
            <STATEVAR type = "Double" key="Requirement1"></STATEVAR>
            <STATEVAR type = "Double" key="Requirement3"></STATEVAR>
            <STATEVAR type = "Double" key="Requirement5"></STATEVAR>
            <STATEVAR type = "Double" key="Requirement7"></STATEVAR>
      </CONSTRAINT>

      <CONSTRAINT
```

```
            value="0.79"
            constraintName = "Constraint2"
            type="FAIL_IF_HIGHER">
            <STATEVAR type = "Double" key="Requirement2"></STATEVAR>
            <STATEVAR type = "Double" key="Requirement4"></STATEVAR>
            <STATEVAR type = "Double" key="Requirement6"></STATEVAR>
            <STATEVAR type = "Double" key="Requirement8"></STATEVAR>
      </CONSTRAINT>

</MODEL>
```

Appendix E – SystemEval Namespace

For the systemEval namespace, request access from Dr. Eric Mehiel at California Polytechnic State University, San Luis Obispo.